Kale: Elastic GPU Scheduling for Online DL Model Training

基本信息

Source：SOCC 2024

Authors：

Ziyang Liu[1], Renyu Yang[1*], Jin Ouyang[2], Weihan Jiang[1], Tianyu Ye[1], Menghao Zhang[1], Sui Huang[2], Jiaming Huang[2], Chengru Song[2], Di Zhang[2], Tianyu Wo[1], Chunming Hu[1]

[1]Beihang University, Beijing, China

[2]Kuaishou Inc., Beijing, China

Introduction

对于许多服务供应商来说，在线模型训练在提供搜索、推荐和广告等互联网规模业务方面变得非常重要。然而，当前离线训练系统中用于弹性资源调度的自动缩放方法在在线模型训练场景下通常会遇到 GPU 资源调整滞后的问题，并可能导致在线模型训练的准确性降低和速度减慢。

离线模型训练：训练数据在训练开始前就确定好了。

在线模型训练：在训练过程中，数据以流式方式输入模型，训练过程持续进行。在线模型根据实时数据更新和调整其参数，从而更好地适应不断变化的用户行为和偏好。

> 有一个在线推荐系统，每当用户浏览一个商品时，系统会立即接收到该行为数据，并根据这个数据实时调整推荐算法，更新推荐内容，而不需要等待用户完成整个浏览过程或一次性收集大量数据。

离线场景的弹性调度机制：通过精确测量训练吞吐量和提前规划资源调整来优化离线训练场景中的作业或集群吞吐量而设计的。

局限性：

- 离线模型训练中采用的弹性调度器对数据流量的感知通常具有滞后性，无法满足在线训练的时效性需求。
- 传统自动扩展策略需要时间才能生效，可能导致在线模型错过大量数据，导致训练结果过时和不准确。

  例如：一个1B参数的在线模型训练，自动伸缩的默认耗时约为10分钟（容器启动30秒，镜像拉取30秒，模型加载6分钟，数据流重建2分钟）

Background

Online DL Model Training

实时流数据传输：在线模型训练通常通过消息平台（例如 Apache Kafka）支持的实时流数据进行管道传输，可以使模型随着新数据的到来而不断更新。

流数据分区：在线数据流被组织并存储在topics中，并对topics进行分区，放置在不同节点中。——数据预处理阶段，根据数据属性（如：年龄、位置）对数据进行划分。

不同工作线程使用不同分区的数据，分配给不同worker 。

Sparse DL Models

搜广推模型变得越来越稀疏。

稀疏模型：大多数参数的值为零，只有少量非零值，例如：**词嵌入（word embeddings）**和**特征嵌入（feature embeddings）**

稠密模型：例如：Transformer（self-attention、Feedforward Layers）、全连接、卷积层

Online Training Performance

性能指标：

throughput：数据到达的吞吐量，表示上游入口数据流的速率（每秒记录数CPS）

lag：传入样本在队列中的停留时间，即数据样本从进入消息队列到被worker用于模型训练的时间

流延迟对在线训练的影响：

数据是实时流入模型的，流延迟会影响模型训练的及时性和准确性

> 点击率预测（CTR）和转化率预测（CVR）反映了用户的点击行为和转化行为，可以用于评估深度推荐和广告系统。AUC（Area Under Curve）是用于评估模型性能的指标，尤其是在排序任务中。AUC越高，表示模型对用户偏好或点击模式的预测越好，相关项目的排名质量越高。

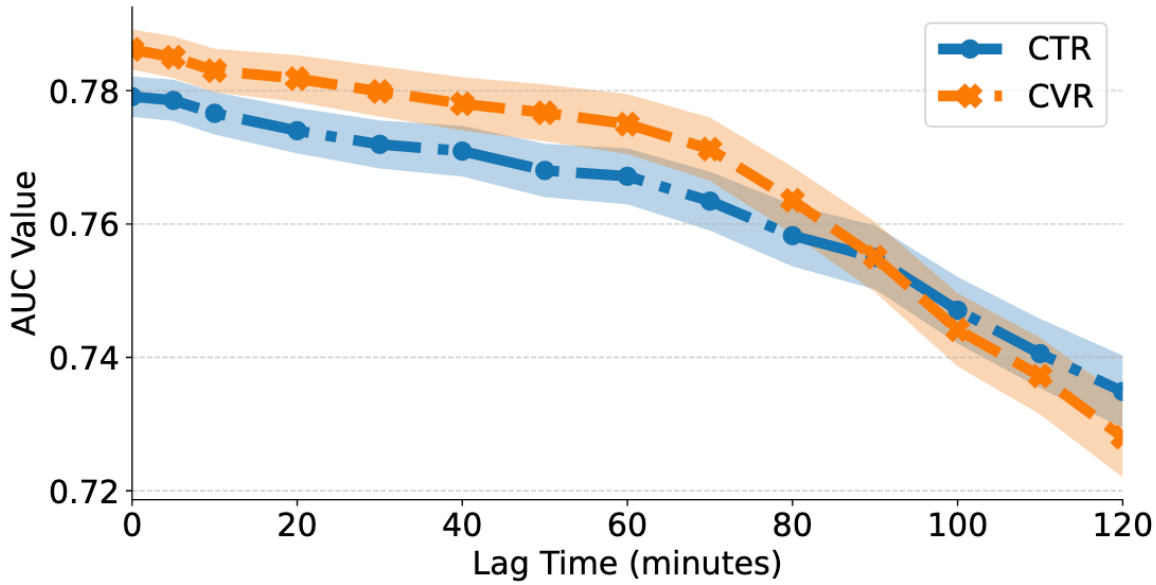
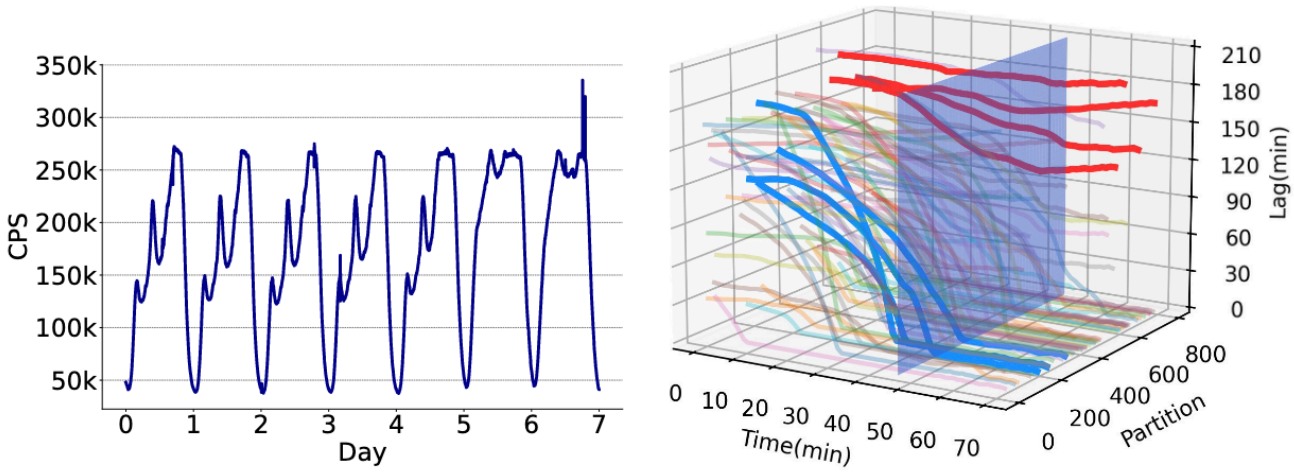


Figure 1: Variation of model AUC with lag time.

结论：随着lag time增加，模型性能变差

Characteristics of Streaming Samples



(a) Temporal distribution (b) Spatial distribution

Figure 2: Uneven spatio-temporal distribution of streaming samples during online learning: a) sample fluctuation over 7 days b) data processing lag of different streaming partitions over time

时间分布不均匀：样本到达率表现出周期性波动和明显的高峰非高峰模式，最大最小差异高达7.65倍，设计动态自动放缩很有必要

空间分布不均匀：训练过程中不同流数据分区的lag不同。有些分区（红色）存在**样本堆积**，训练worker无法足够快地使用上该训练数据；有些分区（蓝色）的worker没有得到充分利用。数据预处理阶段对数据的划分不均匀，使得不同类别之间的用户行为差异较大。

Overview of KALE

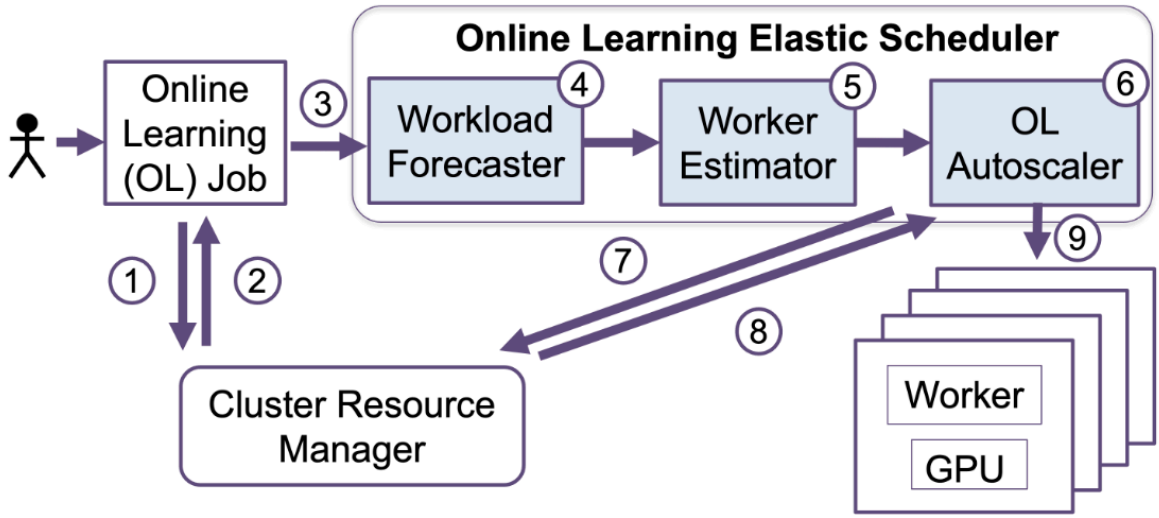


Figure 3: Overview of KALE.

Data Traffic Forecasting

目标：基于历史数据预测即将到来的数据样本吞吐量

方法：采用基于 Transformer 的时间序列预测模型（Fedformer）来学习流数据样本的周期性时间特征，预测即将到来的数据流量。

Resource-Throughput Modeling

关键：确保在给定分配的 GPU 数量的情况下，作业的训练吞吐量（即每秒样本数）应大于传入流量速率。

建模：捕获稀疏深度模型的 **GPU 资源**和**吞吐量**之间的关系

参数服务器架构

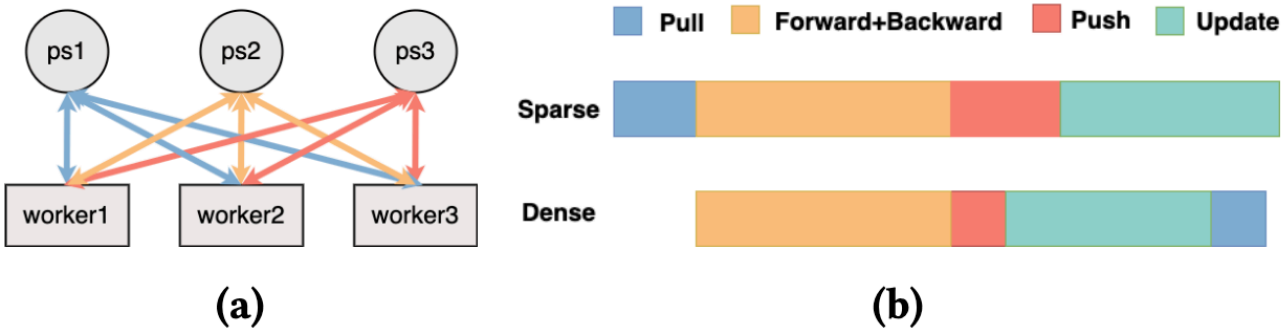


Figure 4: (a) Parameter-server architecture; (b) Pipelines and overlaps of operations for processing sparse and dense parameters

采用PS架构进行数据并行。

更新策略：

关键思想是通过稀疏参数和稠密参数分开存储和处理来优化参数存储和更新策略。

- 密集参数根据计算的梯度完全更新。
- 稀疏参数根据相应的关键值进行更新，只有非零的稀疏参数才会被更新。每次训练迭代中稀疏参数更新的数量与批量大小正相关。

传输：稀疏参数和稠密参数分开传输

- 密集特征的pull时间与稀疏特征的update时间overlap。
- 稀疏特征可以在pull期间独占带宽，从而加速模型训练。

计算训练时间

每个工作节点的一个训练步骤（iteration）的时间消耗：

$$T = T_D + T_F + T_B + T_U + T_O + T_C \tag{1}$$

设参数服务器的数量为p，工作节点的数量为w，每个工作节点训练的batch为m，密集参数的大小为D，稀疏参数的大小为$m \times S$

假设参数服务器上的参数是均匀分布的，则工作节点向参数服务器发送的密集参数的数量为D/p，稀疏参数的数量为$m\times S/p$

$$
\begin{aligned}
T = m \cdot T_{\text{forward}} + T_{\text{backward}} + \frac{2 \cdot m \cdot S + D}{B \cdot p} \\
+ \frac{m \cdot S}{p} \cdot T_{\text{update}} + \lambda \cdot p + \lambda' \cdot w
\end{aligned}
\tag{2}
$$

吞吐量=总批量大小 M / 1 step训练时间

每个worker上的batch数量为M/w

计算同步训练的吞吐量：

$$
\begin{aligned}
f_{\text{sync}}(w,p) = M \cdot \Bigg(& \frac{M}{w} \cdot T_{\text{forward}} + T_{\text{backward}} \\
& + \frac{2 \cdot \frac{M}{w} \cdot S + D}{\min\left\{\frac{B_0}{w}, \frac{B_0}{p}\right\} \cdot p} + \frac{\frac{M}{w} \cdot S}{p} \cdot T_{\text{update}} \\
& + \lambda \cdot p + \lambda' \cdot w \Bigg)^{-1}
\end{aligned}
\tag{3}
$$

将w 和 p 视为主要变量，可将表达式简化为：

$$
f_{\text{sync}}(w,p) = \left(\theta_0 + \frac{\theta_1}{w} + \frac{\theta_2}{p} + \frac{\theta_3 \cdot w}{p} + \frac{\theta_4}{w \cdot p} + \theta_5 \cdot p + \theta_6 \cdot w \right)^{-1}
\tag{4}
$$

如果存在w/p最佳性能比，可简化表达式为：

$$
f_{\text{sync}}(\mathbf{w}) = \left(\theta_0 + \frac{\theta_1}{\mathbf{w}} + \frac{\theta_2}{\mathbf{w}^2} + \theta_3 \cdot \mathbf{w} \right)^{-1}
\tag{5}
$$

计算异步训练的吞吐量：worker数量乘以每个worker的单独吞吐量。

$$
f_{\text{async}}(\mathbf{w}) = \mathbf{w} \cdot \left(\theta_0 + \frac{\theta_1}{\mathbf{w}} + \theta_2 \cdot \mathbf{w} \right)^{-1}
\tag{6}
$$

Autoscaling

核心思路：

- **初始资源分配计划：**在给定的时间间隔内，首先制定一个初步的资源分配计划，该计划为每个时间段分配适当数量的工作节点（workers），以便能够处理预期的数据样本量。这样可以确保每个时间段的计算负载有足够的工作节点来处理

- **避免不必要的自动扩展：**在在线学习和自动弹性扩展过程中，由于数据流量的波动性和度量指标收集的动态性，可能会导致流量吞吐量的波动。如果没有适当的干预，直接按照基本扩展计划进行资源调整，会导致快速且频繁的资源扩展，从而引发系统的震荡或不稳定。

初始化资源分配

目标：确定能够提供足够大的吞吐量来处理流数据样本的最少worker数量

可以定义为在时间点t执行的优化过程：

$$\begin{aligned} \min_{w(t)\in\mathbb{Z}^+} \quad & w(t) \\ \text{s.t.} \quad & \mathcal{F}(w(t)) > \mathcal{L}(t) \end{aligned} \tag{7}$$

$w(t)$表示t时间分配的worker数量，F表示分配的work能够实现的吞吐量（采用上述的拟合模型），$L(t)$表示预测模型在t时间的预测值。

- 在训练过程中保持batch size不变， w/p的比率在缩放过程中保持恒定
- 优先考虑训练吞吐量，而不是GPU效率
- 对推断的 GPU 数量使用舍入策略来保证训练性能
- 不考虑使用共置/多实例 GPU 来提高 GPU 效率
- 不考虑异构

稳定化机制

关键：

由于工作负载波动或短暂的数据爆发所导致的波动并不一定需要资源调整。

算法：

- 初始资源分配：通过初步的资源分配计划来处理每个时间点的资源需求（预测的数据样本量）
- 资源调整检测：两个时间点之间的资源变化超过ρ，算法会检查是否需要进行自动扩展
- 资源持续时间计算：`CalcDuration()`函数计算当前资源需求持续的时间长度，如果该持续时间低于设定的阈值τ（Line 7），则触发校准程序
- 校准程序：校准程序会利用前一个或后一个时间点的资源分配（相邻资源计划之间的最大数量）对当前时间点的资源分配进行校准（Line 8-9）。这样可以平滑短期内的资源波动，避免过于频繁的资源扩展。

参数设置：

ρ（阈值）：ρ是用来判断资源分配变化是否显著的阈值。如果两个时间点之间的资源变化超过ρ，表示需要考虑进行资源调整。较大的ρ会使系统对资源扩展的敏感性降低，从而减少自动扩展次数，但可能会错过及时的调整，影响模型的精度。默认为1

τ（持续时间阈值）：τ是判断资源需求持续时间的阈值。较大的τ意味着如果资源需求维持时间较短，就不会进行自动扩展，从而减少了系统开销。较小的τ会使系统对频繁的变化更加敏感，可能导致更多的自动扩展。较大的τ虽然能减少系统开销，但可能会导致延迟的资源调整，从而影响模型的精度。默认为10分钟

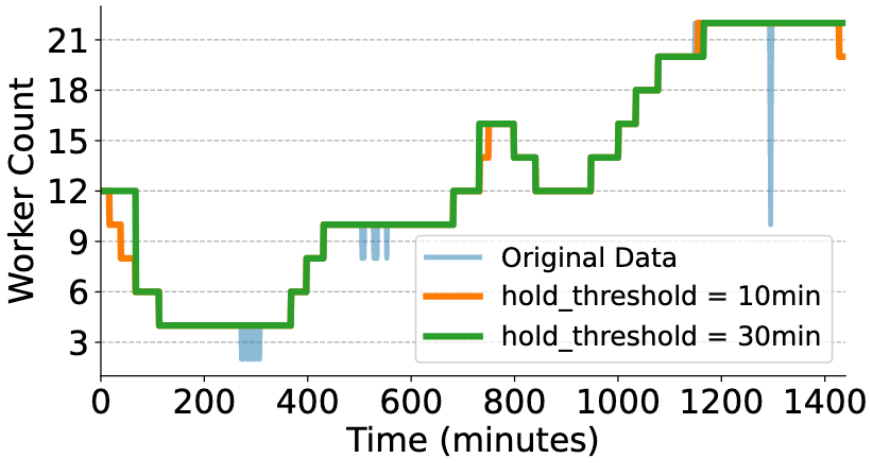


Figure 5: Resource allocation: initial plan vs. stabilized plan

省略了一些由于抖动而引起的资源调整，从而保证了系统的鲁棒性。

Global Data Shuffling

由于数据分布的不平衡性质，进入不同worker的数据流可能会有很大的不同。

全局洗牌机制（global shuffling mechanism），用于在多个工作节点（workers）间高效地分配流数据。在分布式训练或数据处理环境中，数据的分布往往不均衡，因此需要通过特定机制来调整和优化数据的流向，避免数据过载或资源闲置。

1. **Shuffle Unit的角色**：
   - 每个工作节点都配备一个 **Shuffle Unit**，负责管理数据的流动。当一个工作节点的本地队列过满时，Shuffle Unit会将溢出的数据样本转发给其他空闲的工作节点。
   - 转发数据时，Shuffle Unit会根据目标节点的状态和容量进行选择，避免造成进一步的负担。
2. **阈值驱动的转发策略**：
   - 该机制使用**阈值驱动**策略来决定何时以及重定向多少数据样本。例如，当工作节点的队列满了，它会停止接收数据，直到队列长度回落到某个较低的阈值。
   - 数据发送者会在本地队列满时，将后续的样本缓存在缓冲区中，直到可以转发出去。
3. **全局视图与事件驱动的广播**：
   - Kale使用**全局视图**来跟踪每个工作节点的状态，确定哪些节点可以接收溢出的数据。这种方式避免了集中式的协调管理，改为基于事件驱动的通知广播。
   - 每当工作节点的状态发生变化（例如，加入或移除），相关信息会立即广播给其他工作节点，确保所有节点都能获取到最新的状态。
4. **数据传输实现**：
   - 数据通过**bRPC**（一个高性能的网络通信框架）进行传输，序列化后作为字节流发送到目标节点。
   - 在目标节点接收到数据后，数据会被解析并恢复，接着通过流式数据处理框架（如流式RPC）进行实时处理。

总之，该机制通过全局协调和智能的资源管理，确保了流数据在多个工作节点间的高效传递和处理，避免了负载过重的情况，提高了分布式系统的性能和可靠性。

EXPERIMENTS

Experiment Setup

Hardware and Software：

包含32 个 NVIDIA A10 GPU、Intel Xeon Platinum 8352Y CPU @ 2.20GHz 和 1TB RAM 的集群

实时数据流由 Kafka 管理，每个Kafka主题包含800个分区。

Workloads：

排名模型具有大约十亿个参数，并遵循专为复杂推荐或广告任务而设计的复杂架构。输入特征是高维稀疏向量

Evaluation Metrics

SLO violation rate：将超过 20 分钟的滞后视为违反 SLO 的迹象，测量在整个实验时间范围内经历此类 SLO 违规的持续时间的比例

Accumulated lag：训练中一段时间内流数据样本的累积程度，计算为每分钟滞后的总和。

Downtime：由于缩放资源导致的时间开销（包括保存模型、启动新的worker容器、拉取镜像、加载模型、重建..）

GPU hours：在线学习中 GPU 使用的累积持续时间。该指标的计算方法是将 GPU 总数乘以每个 GPU 积极参与任务执行的时间

Effectiveness of Throughput Modeling

硬件配置：NVIDIA A10 和 NVIDIA T4

收集多组数据对（w，f（w）），其中w表示worker数量，f（w）表示模型吞吐量，使用非负最小二乘法（NNLS）进行验证

the global batch size as M = 16, 384

w/p设为1

baseline：

Ernest 2016

Optimus 2018

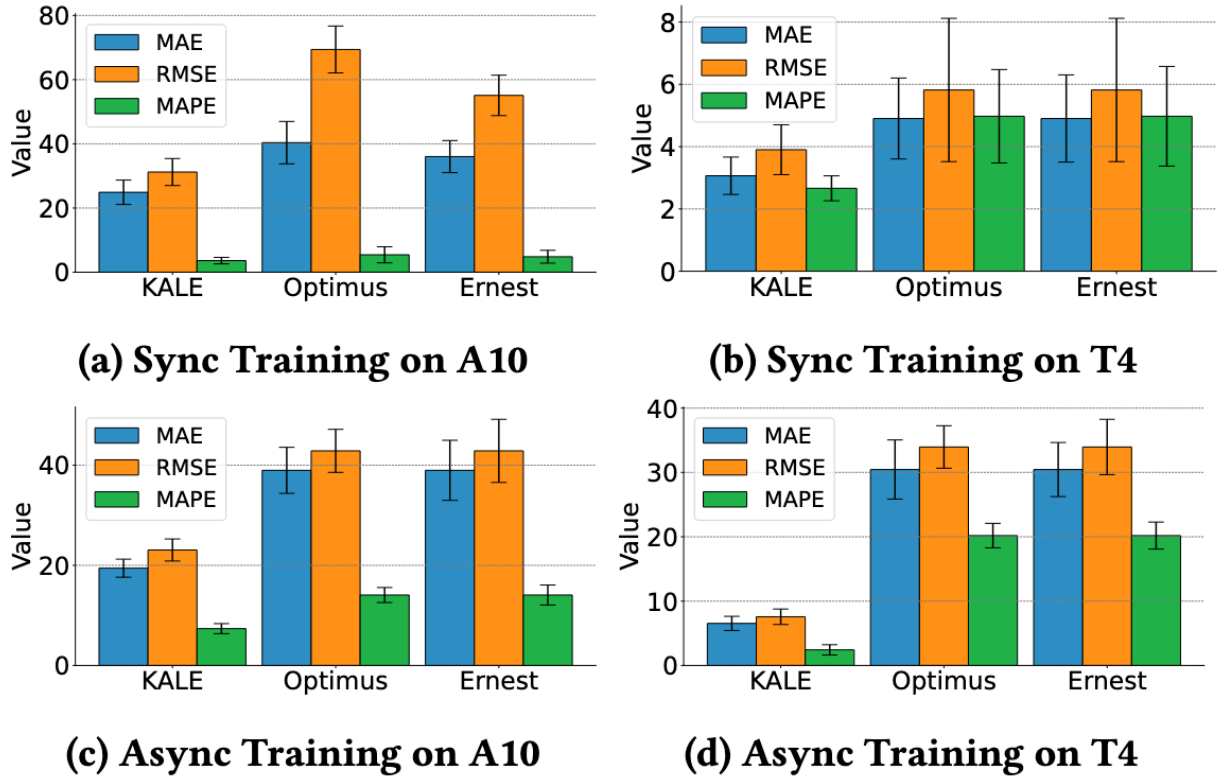


Figure 7: MAE, RMSE and MAPE under different methods.

结果表明当前吞吐量建模机制用于处理 Kale 中波动的数据流的稳健性和有效性。

Effectiveness of Global Shuffling

测量每个分区的延迟、资源消耗（包括网络带宽和 CPU 利用率）以及 AUC

baseline：No Shuffling / Basic Shuffling / Advanced Shuffling:

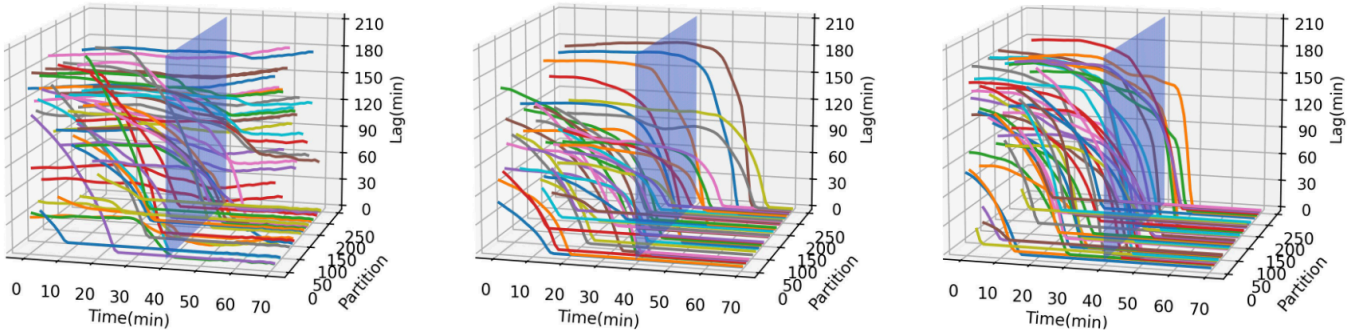


Figure 8: The accumulation of partition samples under different shuffle strategies: (a) No Shuffling; (b) Basic Shuffling; (c) Advanced Shuffling. The vertical axis represents the time from when the samples enter the message queue to when they are read, reflecting the time they are accumulated.

未采用洗牌策略时，部分数据分区出现较长的流延迟，部分分区甚至有延迟增大的趋势。其中高级洗牌通过按需数据转发和分散的状态通知机制，可以比基本洗牌更均匀、更快速地消耗样本 - 所有分区的滞后更快地降至零。

Table 2: Comparison of network traffic, CPU utilization, and model AUC under different shuffling strategies.

Shuffling Strategy	Traffic (MB)	CPU Util. (%)	Model AUC (%)
No Shuffling	905 ± 15	75.5 ± 2.0	95.7 ± 0.5
Basic Shuffling	1097 ± 20	80.0 ± 2.5	96.4 ± 0.3
Advanced Shuffling	948 ± 18	77.0 ± 1.8	96.5 ± 0.4

尽管数据洗牌机制增加了网络开销，但Kale中的高级洗牌机制相比基础洗牌减少了13.6%的网络开销。

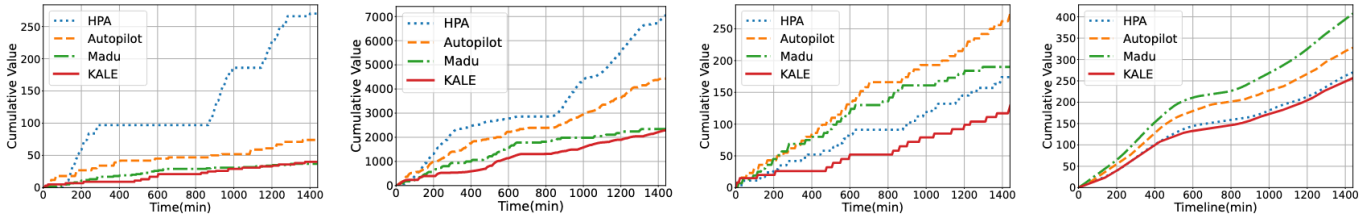
CPU开销仅增加2%，较基础洗牌低3%。

合理的数据转发提高了流量吞吐量，进而提升了模型AUC。

Overall Performance of Kale

Table 3: Comparison of key performance metrics under different scaling schemes.

Scheme	Violation Rate	Accumulated Lag (min)	Downtime (min)	GPU Hours (h)	AUC
Adequate Resources	0	0	0	672	$95.3\% \pm 0.1\%$
HPA	$19.57\% \pm 1.50\%$	$7,151 \pm 150$	163 ± 8	268 ± 15	$95.1\% \pm 0.3\%$
Autopilot	$5.49\% \pm 1.05\%$	$4,420 \pm 100$	269 ± 10	303 ± 10	$95.2\% \pm 0.4\%$
Madu	$2.57\% \pm 0.41\%$	$2,303 \pm 50$	190 ± 10	455 ± 20	$95.3\% \pm 0.3\%$
KALE (proposed)	$2.60\% \pm 0.32\%$	$2,204 \pm 30$	109 ± 7	242 ± 8	$95.8\% \pm 0.2\%$



(a) Violations Distribution (b) Lag Distribution (c) Downtime Distribution (d) GPU Hours Distribution

Figure 9: Comparison of key performance metrics: (a) Violations Distribution; (b) Lag Distribution; (c) Downtime Distribution; (d) GPU Hours Distribution by adopting different autoscaling schemes.

与反应式自动扩展方法（如Autopilot和HPA）相比，Kale通过主动流量预测显著降低了违规率和积累延迟，累计延迟和停机时间分别减少了69.2%和33.1%，SLO违规率从19.57%降至2.6%。与Madu相比，Kale表现出相当的违规率（2.60% vs 2.57%），但累计延迟更短（2,204分钟 vs 2,303分钟），GPU消耗更少，降低了46.8%（242 vs 455）。Kale的资源吞吐量建模和自动扩展校准机制有效减少了不必要的扩展操作，显著减少了停机时间。

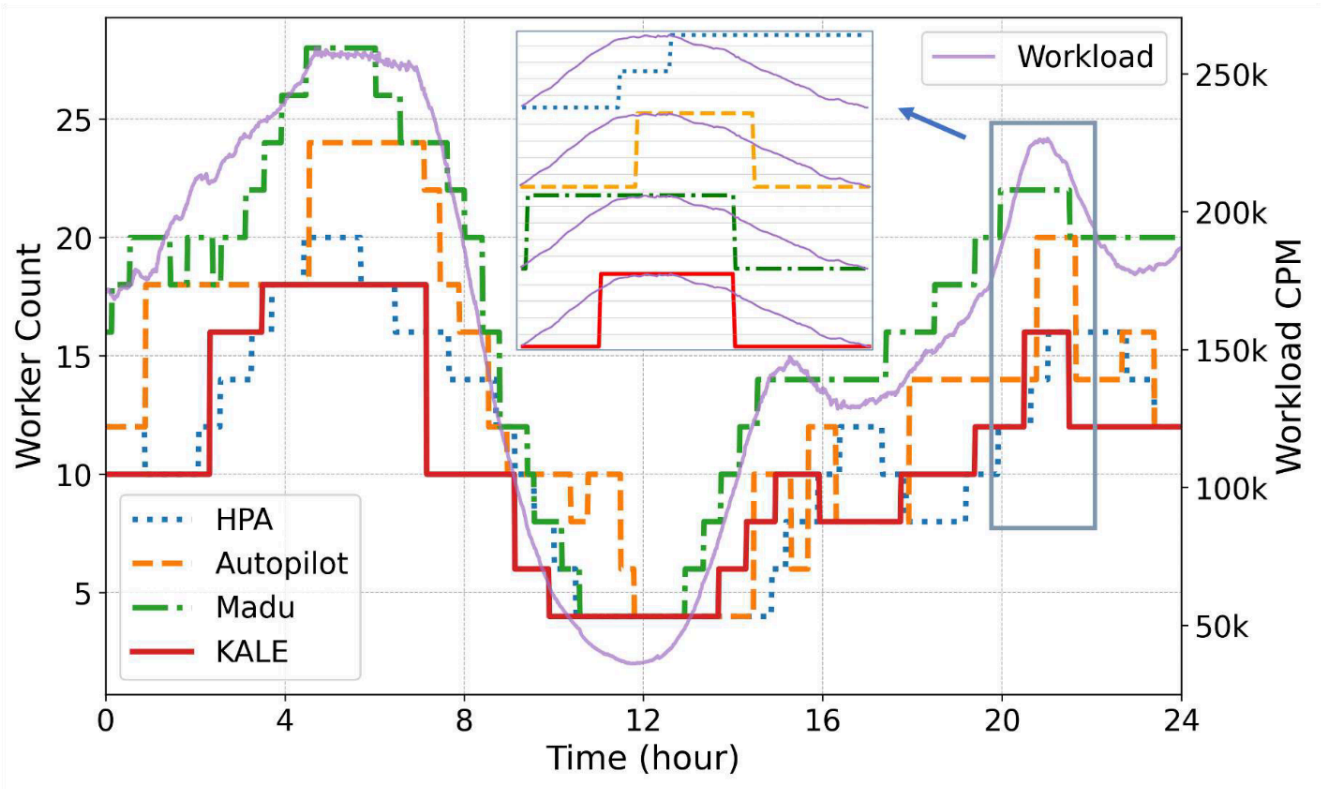


Figure 10: The number of allocated workers by adopting different autoscaling schemes.

HPA（Horizontal Pod Autoscaler）：存在明显的延迟扩展问题，工作负载上升后20分钟才增加工作节点，导致数据积压加剧，流延迟更长，违反率和积累延迟最高。

Autopilot：GPU分配与工作负载高峰略有错位，导致样本积累和流延迟增加，但比HPA表现稍好。

Madu：在高峰期过度且长时间分配资源，导致GPU小时数较高，但未能有效优化流延迟。

Kale：具有更平滑的自动扩展过程，避免了过于频繁的扩展，能够更好地应对负载波动，减少样本积累和延迟。